Applications of Linear Types

A Part III project proposal

D. C. Makwana (*dcm41*), Trinity College

Project Supervisor: Dr. N. K. Krishnaswami, Dr S. Dolan

Abstract

Complex resource management is a challenge for programmers. Linear types allow the compiler and the user to statically keep track of the resources that an implementation uses, and offer a promising solution to resource management. However, they have not made their way into mainstream programming languages. I aim to take a step towards this with an incremental approach, by allowing users to use linear types via an OCaml library that offers them the ability to manipulate primitives of the Owl numerical library in a safe and efficient manner, by inferring and removing redundant copies.

1 Introduction, approach and outcomes

Simply stated, a linear value is a value that must be used once and only once. More abstractly, it is the type-theory corresponding to Girard's Linear Logic. Examples of linear types in the real world are the recent efforts to extend GHC with Linear Types [3] for safe, pure-functional streaming: a pointer into a stream must be processed before moving onto the next element (one use) but crucially, you cannot keep this pointer and refer to it again (only one use). Rust implements linear types behind the scenes but presents *affine* types to the programmer: if a destructor is not called explicitly, the compiler inserts ones when it infers the value's lifetime has ended. Apart from memory (which enables zero-copy buffers and state-management in a pure functional language), linear types can also be used to control other resources such as files, network communications, pointers with capabilities and concurrent channels.

My project will focus on memory, but not manipulating raw memory block primitives, but ensuring the primitive types (such as large matrices) that Owl [4] manipulates are used efficiently. Owl provides an immutable interface and so performs several redundant copies during a computation. It does provide more efficient implementations of common numeric algorithms as primitives which are hand-optimised and abstract away any unsafe calls involved. This approach trades performance for conciseness, safety and ease of reasoning.

The opposite approach – to rely on the programmer to note and remove redundant copies – is used extensively by Fortran's BLAS [1] and Boost's uBLAS [2]. They offer many different conventions for each operation to meet every situation. For example, a function implementing a binary operation on two matrices may return a newly-allocated matrix containing the result or modify one of two or three input arguments to hold the result (if there is no aliasing, like in `noalias(C) = prod(A, B)` where the `=` operator is

overloaded) and even specify temporary intermediate results with annotations (as with `prod(A, prod<temp_type>(B,C))`).

I hypothesise that it is possible to have the best of both worlds: write your code as you wish and then have the computer statically determine the most space-efficient sequence of copies and in-place writes that produces the desired result. A small example to illuminate the point follows. Suppose we have $n \times n$ matrices A, B and C. Knowing all those matrices are used once and only once, we would then infer that we can evaluate the expression `(A + B) + C` *in-place*, say, using the memory that A occupies.

```
let y  = (A + B) + C in
(* is the same as *)
let i1 = A + B  in
let i2 = i1 + C in
i2
(* is compiled to *)
A := A + B;
A := A + C;
return A
```

```
let y  = ((A + B) + C) * A in
 (* is the same as *)
let (A1, A2) = dup(A) in
let i1 = A1 + B       in
let i2 = i1 + C       in
 let i4 = i2 * A2      in
i4
(* is compiled to *)
B := B + A; B := B + C;
B := B * A; return B
```

However, when an expression is used more than once, we can be more explicit and invoke a function *dup* for duplicate, to save A for use later.

It is important to note that *dup* is not necessarily a copy: it is merely a signal to the compiler via the type system that A's memory should not be written to until both the results of the *dup* (A1 and A2) are used. Indeed, knowing that B is not used again, we could store (intermediate) results in its memory.

2 Workplan

The output of this project will be a library that will, given a linear algebra expression, produce OCaml code to compute the expression using mutability and unsafe functions that will, thanks the static guarantees provided by linear types, be safe, correct and perform well (less memory consumption, fewer dynamic checks). Evaluation will consist of comparing test programs against Owl's default immutable interface, its lazy interface, Julia and Python/Numpy.

During implementation, I will also explore other questions such as: if a matrix does need to be copied, then, when during execution is it best to do so? What is necessary to support the class of numerical algorithms like the small example above, such as Kalman filters that although in their mathematical expression appear to use a value twice, can be implemented to have values modified in-place? Numerical libraries and lazy interfaces (that use mutability behind the scenes by reference counting) tend to perform poorly for allocating lots of small matrices (for example, in robust linear regression); will statically determining

memory usage improve their performance?

Date	Work
04-12-2017	Set-up build environments (using Nix) Git repository continuous-integration and testing framework. Start designing explicitly-typed DSL (domain-specific language).
18-12-2017	Finish designing DSL and write a tree-interpreter for it that outputs matrices using Owl.
01-01-2018	Write a lexer/parser and *compiler* for this language.
15-01-2018	Review with supervisor/break for assignments.
29-01-2018	Start with basic elaboration and inference.
12-02-2018	Buffer.
26-02-2018	Project progress review. Begin discussion of advanced extensions such as staging or further inference and analysis.
12-03-2018	'One-minute madness: project 'elevator pitch". Start with chosen extension. Submit to ICFP.
26-03-2018	Plan evaluation and start and write-up.
09-04-2018	Buffer.
23-04-2018	Perform evaluation.
07-05-2018	Write-up.
21-05-2018	Project title changes (25^{th}). Final changes to code and thesis. Prepare for POPL.
01-06-2018	Submission Deadline.
12-06-2018	MPhil and Part III Project presentations.

References

[1] Blas (basic linear algebra subprograms). http://www.netlib.org/blas/. Accessed: 20/11/2017.

[2] Boost ublas. http://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas/doc/operations_overview.html. Accessed: 20/11/2017.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Retrofitting linear types, 2017.

[4] Liang Wang. Owl - an ocaml numerical library. https://github.com/ryanrhymes/owl. Accessed: 20/11/2017.